A Hardware Acceleration Unit for MPI Queue Processing

Ron Brightwell, K. Scott Hemmert, Richard Murphy, Arun Rodrigues,
and Keith D. Underwood
Sandia National Laboratories†
P.O. Box 5800
MS-1110
Albuquerque, NM 87185-1110
Email: {rbbrigh, kshemme, rcmurph, afrodri, kdunder}@sandia.gov

***Abstract*— With the heavy reliance of modern scientific applications upon the MPI Standard, it has become critical for the implementation of MPI to be as capable and as fast as possible. This has led some of the fastest modern networks to introduce the capability to offload aspects of MPI processing to an embedded processor on the network interface. With this important capability has come significant performance implications. Most notably, the time to process long queues of posted receives or unexpected messages is substantially longer on embedded processors. This paper presents an associative list matching structure to accelerate the processing of moderate length queues in MPI. Simulations are used to compare the performance of an embedded processor augmented with this capability to a baseline (embedded processor only) implementation. The proposed enhancement significantly reduces latency when queues grow to moderate length while adding virtually no overhead for extremely short queues.**

I. Introduction

In the mid-1990's, message passing became the dominant mechanism for programming massively parallel processor systems. By the late-1990's, the majority of message passing programs leveraged the MPI Standard [1]. In the intervening years, billions of dollars have been invested in developing application codes using MPI. Thus, it has become critically important to insure that new systems implement MPI as efficiently as possible.

Many approaches have been taken to characterizing the efficiency of MPI. The most common (and least useful) is to simply evaluate the ping-pong latency and bandwidth of the network. While these are necessary first order measures, models such as LogP [2] (and the LogGP extension [3]) are much more useful. Early work with these models [4] indicated that the most important thing for applications was to minimize the overhead (defined as the time the application processor was involved in starting or finishing the communication). As a result, some of the highest performing networks have chosen to offload much of the work of sending and receiving MPI messages onto the network interfaces [5], [6], [7].

Unfortunately, the second largest impact on application performance is gap (effectively, the inverse of the message rate). Recent work [8], [9] has indicated that applications tend to traverse a significant number of entries in the two primary queues managed by MPI: the posted receive queue and the unexpected message queue. For networks that use embedded processors to traverse these queues, time spent traversing queues leads to an increase in gap. Thus, a compromise has been made to decrease overhead while sacrificing an increased gap in some scenarios.

This paper proposes a unique hardware structure to augment a traditional embedded microprocessor to accelerate list traversal and matching. The proposed hardware uses associative matching structures similar in concept to those found in ternary content addressable memories (TCAMs) to perform high-performance parallel matching. These structures are augmented with list management capabilities to sup-

† Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

port the unique combination of ordering semantics and high list entry turnover needed to support MPI point-to-point message passing semantics.

To better understand basic properties of the design, a prototype has been created in FPGA hardware. The prototype provides an idea of both the clock rates that can be achieved and the timing that should be expected. It also serves as an avenue to explore and refine issues with the control interface. Unfortunately, this implementation would be difficult to integrate into a "real" environment. Thus, system-level simulation was used to demonstrate the usefulness of the proposed hardware. An MPI implementation was created that leverages the hardware acceleration unit. Using simulation, this MPI implementation was compared to a baseline implementation using only an embedded processor with the same benchmarks discussed in [10].

The following section provides further background information on the semantics of MPI and discusses other related research activities. The software interface and hardware design are described in Sections IV and III, respectively. The MPI implementations and simulator are described in Section V. The results from the comparison are presented in Section VI and conclusions are presented in Section VII.

II. Background

Conceptually, an MPI implementation has two message queues — one that contains a list of outstanding receive requests (the posted receive queue) and one that contains a list of messages that have arrived that do not match any previously posted requests (the early arrival or unexpected queue). Incoming messages traverse the posted receive queue for a possible match and end up in the unexpected queue if no match is found. Before a request can be added to the posted receive queue, the unexpected queue must be searched for a possible match. The search of the unexpected queue to add an entry to the posted receive queue must be an atomic operation to insure that a matching message does not arrive between the time the unexpected queue is searched and the receive is posted.

MPI messages are matched using three fields: context identifier, source rank, and message tag. The context identifier represents an MPI communicator object. This system-assigned message tag provides a safe message passing context so that messages from one context do not interfere with messages from other contexts. The source rank represents the local rank of the sending process within the communicator, and the message tag is a user-assigned value that can be used for further message selection within a particular context. A posted receive must explicitly match the context identifier, but may "wildcard" the source rank and message tag values to match against any value. In addition to the matching criteria, MPI also mandates an ordering constraint. Messages between two nodes in the same context must arrive in the order in which the corresponding sends were initiated.

All of the MPI implementations described in published literature represent the posted receive and unexpected queues as linear lists (all of these implementations are based on either MPICH [11] LAM [12], MPI/Pro [13], MPICH2 [14] or LA-MPI [15]). Using this method, the time to traverse these queues grows linearly with the length of the list [10]. And, the length of the list can grow linearly with the number of processes in the parallel application [8], [9]. For some networks, the time spent traversing an arbitrarily long queue may impact the entire system, since the network interface may be unable to service any other requests during the search. This can lead to a situation where a poorly written or erroneous application can affect the performance of other applications in the system.

In order to reduce the search cost, approaches using hash tables have been explored [7], [16]. Hash tables can significantly reduce the time needed to find a matching entry, but can also significantly increase the time needed to insert an entry into the list. Unfortunately, this increase in insertion time has been prohibitive. The increase in insertion time for a hash relative to a list is especially noticeable in the zero-length ping-pong latency test by which high-performance networks are judged. Hashing is also complicated by the need to support wildcard matching and maintain ordering semantics. Unfortunately, an MPI implementation has no *a priori* knowledge of whether wildcard values will be used, so no application-specific approach can be taken.

The use of wildcard matching appears to be widespread. An initial analysis of several applica-

tions at Sandia has revealed that a large number use wildcards. The use of `MPI_ANY_SOURCE`, where the source of the incoming message is not known, is most prevalent. The use of `MPI_ANY_TAG` rarely occurs, perhaps since message tags are intended to be used for differentiating between specific types of messages. Re-coding applications to eliminate the use of source wildcards is non-trivial. The semantic equivalent is to post a receive from every possible source and then cancel those receives that are unused. This strategy is an inefficient use of processing and memory resources.

In this paper, we propose a hardware-based scheme for a network interface to accelerate MPI matching. Previous work has explored approaches to using network interface hardware specifically for MPI. The Quadrics QsNet [6] network has a general-purpose processor on the network interface that allows for running a user-context thread to process incoming messages. This approach allows much of the protocol processing needed to support MPI to occur on the network interface. However, the thread that implements MPI implements queues as linear lists. The network interface for the Sandia/Cray Red Storm machine [5] implements the Portals [17] programming interface, which provides protocol building blocks that support general network functionality as well as MPI efficiently. However, Portals only allows for incoming messages to traverse a linear list and there is no specific hardware to accelerate matching. There is also a significant amount of previous work on using the network interface to implement MPI collective operations efficiently [18], [19], [20]. Similarly, these approaches focus on protocol optimizations and efficient data movement operations rather than list traversal.

The hardware acceleration that we explore in this work is closely associated with the techniques used to accelerate lookups in Internet Protocol (IP) routers. IP routers need to efficiently solve the *longest prefix match* (LPM) problem, where an incoming packet needs to be routed to the network that most closely matches its destination address. As with wildcard values in MPI, network masks can be used to cover an entire range of addresses. For an IP router, an incoming message generates a table lookup for the closest matching destination, ultimately resulting in the selection of an outgoing port. For MPI, an incoming message causes a table lookup on the closest matching posted receive, resulting in the selection of a destination buffer for the message.

A variety of software and hardware approaches have been explored to allow for quickly solving the LPM problem (see [21] for a summary), but none of them are appropriate for MPI for a number of reasons. First, IP routers are expected to be updated infrequently, so updating the list of possible destinations can be a heavyweight operation. The impact of this table maintenance is seen simply as a "hiccup" in the network. For MPI, each posted receive operation causes the table to be updated, so the frequency of table updates is much much higher. Secondly, there is no need to preserve ordering in a router for IP packets. Packets will be routed to the best matching network regardless of when the destination was added to the table. For MPI, the best match may not be the correct match when ordering is considered. If an entry with a wildcard source value is added before an entry with an explicit source, the most exact match should not be chosen. Additionally, the LPM problem assumes a fixed ordering of bits where wildcards only occur at the tail of the address being matched. This is, if any portion of the address is wildcarded, all lower order portions of the address are also wildcarded. Given any fixed ordering of the $\{$*context, source, tag*$\}$ triplet, a field could be wildcarded in the middle without lower order fields being wildcarded. Finally, while the strategies for routing incoming IP packets closely resemble those needed for matching MPI posted receives, they do no match very well with what is needed to handle unexpected message lookups. Unexpected messages actually require a reverse lookup, where a receive to be posted may contain wildcard values that need to be compared to a list of messages with explicit criteria that have already arrived.

III. Hardware Acceleration Unit

The proposed integration of the hardware in the overall network interface chip (NIC) architecture is

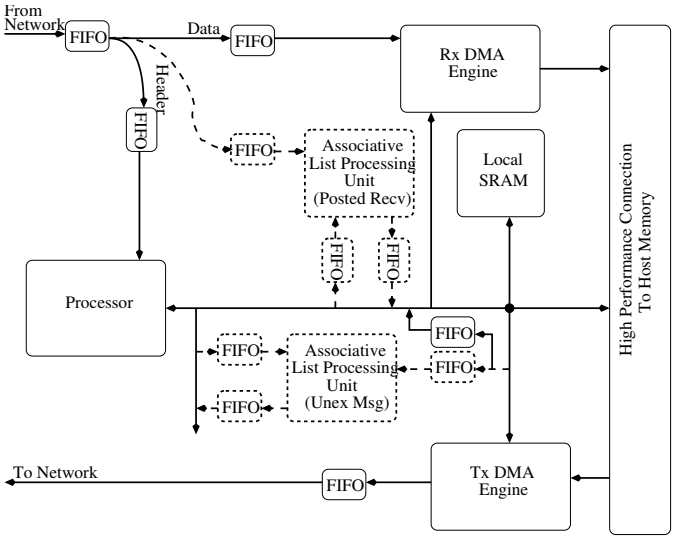


Fig. 1. The proposed NIC architecture leveraging an associative list matching unit (new features shown in dashed lines)

shown in Figure 1[1]. A typical network interface has send and receive (Tx and Rx) DMA capabilities, shown logically separate here, coupled to the network through logical FIFO interfaces that provide a buffering capability. A processor with a local SRAM manages transactions to and from the network and various other housekeeping tasks through a local bus. The proposed new components are shown with dashed lines. To accelerate the posted receive queue, copies of the header information are provided to the associative list processing unit (ALPU) through an added FIFO. A separate command and result FIFO are also provided to enable decoupled, asynchronous interactions with the processor. Similarly, copies of new receives being posted are fed to an ALPU that handles unexpected messages. The details of the ALPU are shown in Figure 2. The hardware is broken into three levels of hierarchy: the individual cell, a block of cells, and the overall associative matching unit.

A. Basic Matching Cell

At the lowest level, one of two individual cells of the match unit is used. A single cell of the match unit for the posted receive queue is shown in Figure 2(a). The cell contains storage for both the set of bits being matched (the MPI matching information) as well as a corresponding set of mask bits (for wildcard bits within MPI). The set of match bits can range from a pair of bits (one each for the two fields in an `MPI_Irecv` that can be wildcarded) to a full width mask as is needed by the Portals interface [17], [22], [23]. In addition, a valid bit, indicating if the entry is valid, and a tag field, used at the discretion of the software, are stored. In the implementation used here, the tag value is a 20-bit pointer into the local RAM that points directly to the matching entry.

To provide matching for the unexpected message queue, the cell is changed slightly, as shown in Figure 2(b). Instead of storing the mask bits in each cell, the mask bits are inputs. In all other respects, the cells are the same. Stored data is passed from one cell to the next. Compare logic (factoring in a set of mask bits that indicate "don't care" locations) produces a single match bit. The basic cell then has three additional outputs that feed into the higher level block. The first is a single bit that is the logical AND of the match bit and valid bit (invalid data cannot produce a valid match). The second is the tag which is muxed through priority logic to select the right match. The final output is a valid bit to allow the higher level block to manage flow control.

B. Block of Cells

At the next higher level (Figure 2(c)), a group of cells is combined into a cell block. In addition to a set of cells, the cell block contains a registered version of the incoming request (to facilitate timing), logic to control the flow of data, logic to correctly prioritize the tags, and logic to generate a "match location". The control flow logic drives a separate enable signal to each cell. The transfer of data from one cell to the next is enabled in two scenarios: when a match occurs and when new items are being inserted. On a successful match, MPI semantics require that the matched item be deleted; thus, the match location is broadcast to all of the cell blocks. Cells at, and below, the match location are enabled while cells above it are not, effectively deleting the matched cell and leaving the lowest priority cell empty.

During inserts, all cells are enabled if there is space available above them to compact any possible

[1]The prototype design only supports hardware acceleration for a single process, but extending it to support a limited number of processes is straightforward.

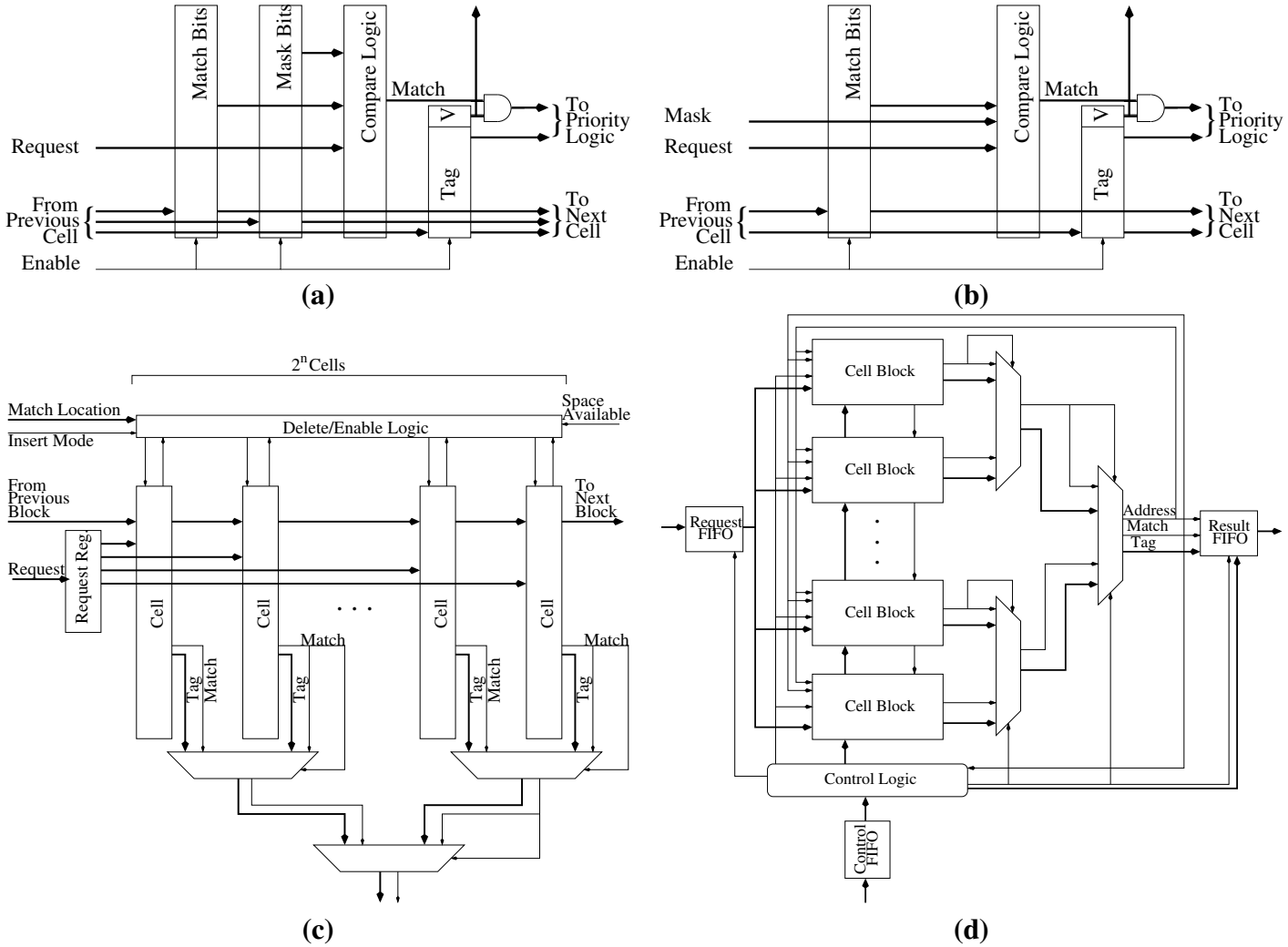


Fig. 2. **(a)** A cell containing a single match unit for the posted receive queue; **(b)** A cell containing a single match unit for the unexpected message queue; **(c)** A block of cells; **(d)** The associative match engine

holes[2]. "Space available" is loosely defined. In the implementation discussed here, "space available" means that either a higher cell in the current block or the lowest cell in the next block is empty. This is to maximize clock frequency in the FPGA prototype and is likely sufficient for all real cases. "Space available" could easily be expanded to include any cell in the next higher block or any cell in any higher block if timing constraints permitted.

The number of cells in a cell block is restricted to a power of 2 to simplify the task of prioritizing the correct tag and generating a correct match location. The prioritization logic uses the match signal to select the "correct" tag for output. In Figure 2(c), the highest order cell (furthest to the right) is the highest priority. In explanation, MPI semantics require that the first matching item in the list be considered the "correct match". In the associative matching structure, list items are inserted from the left and progress to the right. At the first level of prioritization, the higher cell in each pair of cells selects its tag if it matched and the partner tag if it did not. The match bits are also encoded as the lowest order bit of the "match location". At the second level, the logical OR of the highest order pair of match bits forms the select line for the mux and is encoded as the second lowest order

[2]Holes can occur during inserts if there is time between new elements being inserted. Holes do not occur on deletion because all data below the deletion point is shifted upward as part of the delete.

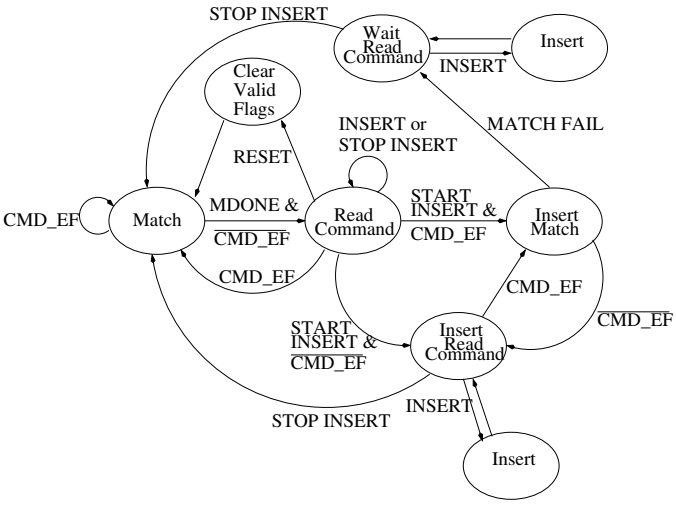


Fig. 3. The controlling state machine

bit of the match location (not shown in figure). This pattern continues through N levels of muxing for 2^N cells. The result is output as the highest order matching tag along with the encoded match location. Obviously, this muxing structure could easily be collapsed with 4-to-1 or even larger muxes, but 2-to-1 muxes improved placement regularity in the FPGA prototype.

C. Associative List Processing Unit

An associative list processing unit (ALPU) chains several cell blocks together and adds control logic to interface to the rest of the network interface. The cell block outputs are combined and prioritized in the same manner as cell outputs are combined in the cell block. Effectively, several cell blocks are combined to create one large, virtual array of cells. The modularization into cell blocks simplifies timing (particularly for the compaction logic) and simplifies the exploration of the design space.

The control logic in the highest level controls the interaction with the rest of the NIC. This control logic determines when new data is taken from the header input, when matches begin, when data is written to the output, when data is read from the control input, and how much space is available in the ALPU. The governing state machine is shown in Figure 3. The state machine begins in the `Match` state. In the `Match` state, the ALPU accepts a new match each time a match completes. Successful or failed matches are output to the result FIFO. If a new command arrives (the command FIFO becomes not empty), then at the completion of the current match, the state machine enters the `Read Command` state. At this point, only the `RESET` and `START INSERT` commands are valid [3]. A `RESET` clears all of the valid flags and returns to the matching state. A `START INSERT` puts the device into insert mode. Insert mode implies a change in the matching behavior — insert commands are accepted and matching continues until a match fails. Matches are stopped temporarily for each insert (to maintain correctness), but it is likely that the processor cannot fill the command FIFO as quickly as the ALPU can drain it; thus, between inserts, matches are allowed to continue. Successful matches are output to the result FIFO and failed matches are held for a retry. This is described in further detail in Section IV. A `STOP INSERT` command returns the operation to the standard match mode.

IV. Software Interface

Referencing Figure 1 provides insight into how the proposed hardware fits into the overall software architecture. In a traditional NIC (e.g. the Red Storm system), the header and data are separated (logically, if not physically). The processor performs list matching functions using the header information and instructs the DMA. With the proposed hardware, header data would be replicated to the associative list processing unit. The purpose of the ALPU is to quickly provide an index into the match list if a match occurs. If a match does not occur, the processor needs to decide what to do with the non-matching message (described later); thus, the processor also receives a copy of all of the header information. In general, the purpose of each of the FIFOs is to provide hardware level decoupling to enable asynchronous operation.

A. Processor Interface

The ALPU requires a very limited set of commands (see Table I). A pair of commands (`START INSERT` and `STOP INSERT`) are used to instruct the ALPU to enter and exit insert mode (the mode that is safe for inserts). `RESET` is used to clear the ALPU and `INSERT` is used to insert new items.

[3]Other commands are discarded and an empty command FIFO before a valid command causes a transition back to the match state.

TABLE I

ASSOCIATIVE LIST PROCESSING UNIT COMMAND SET

Command	Description	Inputs
START INSERT	Instruct the ALPU to enter insert mode	None
INSERT	Insert a new entry in the ALPU	Match bits, Mask bits (optional), and tag
STOP INSERT	Instruct the ALPU to exit insert mode	None
RESET	Clear all entries in the ALPU	None

Only the `INSERT` has parameters: the match bits to be used, the mask bits if needed, and a user defined tag.

The responses expected from the ALPU are shown in Table II. The `START ACKNOWLEDGE` is returned in the response to a `START INSERT` command and indicates the number of free slots in the ALPU. `MATCH SUCCESS` and `MATCH FAILURE` are the responses that are expected in normal ALPU operation. General operation of the device proceeds as follows. A `START INSERT` and its response (`START ACKNOWLEDGE`) must occur before an `INSERT` can be performed. Inserts may then be performed until a `STOP INSERT`. `MATCH SUCCESS` can occur at any time, but `MATCH FAILURE` cannot occur between a `START ACKNOWLEDGE` and a `STOP INSERT`.

B. Overall List Management

To manage MPI queues using an ALPU, a microprocessor must develop an appropriate set of heuristics. As previously shown [8], [9], these queues can grow to tens or hundreds of items; however, at times, the queues can also be quite short. Because using the ALPU will incur a certain amount of overhead, the software must only use it when the queue is adequately long. In addition, the software must recognize that inserting elements into the ALPU requires a certain overhead and should attempt to conglomerate insertions into the list.

Even though the ALPU will be used for automated high-performance matching, the processor should maintain a copy of each list. The copy of the list allows the ALPU to return a simple pointer to a list entry, instead of the entire entry. As entries are matched (and, thus, deleted in the ALPU), the processor's copy of the entry must also be deleted. Furthermore, the processor may have entries that have not yet been entered into the ALPU. A pointer to the start of the portion of the list that has not been entered into the ALPU should also be maintained for proper handling of responses (discussed further in Section IV-D).

C. Match Entry Insertion

When the hardware is initialized, the ALPU is empty. Matching is enabled, but no matches will succeed; thus, the hardware should be designed such that the processor can disable the delivery of duplicate information (headers or new posted receives) to the ALPU until it is initialized. As new entries for the queue arrive (new posted receives or unexpected messages), the processor should begin to build the appropriate queue in memory. When the queue length crosses a threshold (defined by heuristics to enable the best overall performance), the processor sends a `START INSERT` command to the ALPU. To avoid a potential race condition where the a match in the pipeline fails while the processor is performing an insert[4], the processor must wait for an `INSERT ACKNOWLEDGE` response. In response to the `START INSERT` command, the ALPU enters a safe state where matches can occur, but matches that fail are held for retry until after all inserts complete.

While waiting for the `INSERT ACKNOWLEDGE`, the processor may receive one or more `MATCH SUCCESS` or `MATCH FAILED` responses. These must be handled correctly as described in the next section. The `INSERT ACKNOWLEDGE` will include a field indicating to the processor the number of entries it is safe to insert. In an optimal implementation, the processor will also track this number to insure that it does not attempt to start inserting when little or no space is available. Having received the `INSERT ACKNOWLEDGE`, the processor should

[4]Inserts are irrevocable.

TABLE II
ASSOCIATIVE LIST PROCESSING UNIT RESPONSES

Response	Description	Outputs
START ACKNOWLEDGE	ALPU has entered insert mode	Number of free entries
MATCH SUCCESS	Input matched list item	Tag from list item matched
MATCH FAILURE	Input did not match list item	None

perform the desired number of inserts as quickly as possible and then send a `STOP INSERT` command. If the number of inserts to be performed is large, the processor may need to periodically clear the result FIFO of successful matches that occur during the insert process to prevent it from filling. Each insert includes the information to be matched, optionally a set of mask bits (for posted receives), and a tag. The tag can be any value and will be returned on a successful match; however, the recommended use is to store a pointer to the position in local RAM where the corresponding queue entry is stored.

D. Result Handling

If the ALPU is in use, the processor must retrieve a response from the ALPU for every header that is received. Thus, the processor should first retrieve the copy of the data provided to it and then retrieve the response. The response will either be a `MATCH SUCCESS` or a `MATCH FAILED`. On a success, the returned tag can be used to point directly to the matching list item in the processor's copy of the list. On a failed match, the processor must use the local copy of the data and search the portion of the list that has not yet been loaded into the ALPU. If there is still no match, the data must be handled correctly. If the data is a header that did not match an item in the posted receive queue, it should be inserted into the unexpected message queue. If the data is a new posted receive that did not match an item in the unexpected message queue, it needs to be added to the posted receive queue.

V. METHODOLOGY

There were four aspects of this research. The first was a benchmark that exposed a significant problem on modern network interfaces cards (NICs) that leverage embedded processors. The behavior was replicated using a simulation environment that reproduces a modern system environment and provides a platform for research into potential NIC improvements. The simulated NIC was enhanced with the proposed associative list processing unit (ALPU) and the MPI implementation was modified to leverage the feature. Finally, an independent hardware prototype was created to provide insight into the performance of the proposed design.

A. Benchmarks

The primary motivation for this design was to reduce the latency of messages when long posted receive queues or long unexpected message queues were present. The magnitude of the problem was revealed in an earlier study [10] using two newly designed benchmarks. These benchmarks are used again here to study the impacts of the associative list processing unit.

The benchmark designed to measure the impact of changes in the pre-posted receive queue length provides three degrees of freedom: the length of the pre-posted receive queue, the portion of the pre-posted receive queue that is traversed, and the size of the message. This enables the user to measure the impacts of both the receive queue length and the impact of actual queue traversal.

The benchmark created to assess the impact of unexpected message queue length on message latency only allows the length of the unexpected message queue and the size of the message to be varied. It deviates from the traditional way of measuring latency in that it includes the time to post the receive for the latency measuring message as part of the latency. This better reflects the way that MPI is actually used by applications, which typically have some number of iterations and post receives in each iteration.

B. Simulation Environment

System-level simulation of the matching structure used a simulator based on Enkidu [24], a component-based discrete event simulation framework. To simulate the CPU and NIC processors, `sim-outorder` from the SimpleScalar [25] tool suite was integrated into this framework. Components representing a simple network, DMA engines, a memory controller, and DRAM chips were added. The memory hierarchy was modeled to include contention for open rows on the DRAM chips.

The main processor was parameterized to be similar to a modern high-performance processor, such as an AMD Opteron. The NIC processor was parameterized to be similar to a processor in higher end network cards, such as the PowerPC 440 (see table III). A simple bus on the NIC connected the main processor with the DMA engine, SRAM, and matching structure. This bus was simulated with a 20ns delay.

TABLE III
PROCESSOR SIMULATION PARAMETERS

Parameter	CPU	NIC Processor
Fetch Q	4	2
Issue Width	8	4
Commit Width	4	4
RUU Size	64	16
Integer Units	4	2
Memory Ports	3	1
L1 Caches	64K 2-way	32K 64-way
L2 Cache	512K	none
Clock Speed	2Ghz	500Mhz
Lat. To Main Memory	85-90 cycles	30-32 cycles
ISA	PowerPC	PowerPC
Network Wire Lat.	200 ns	

C. MPI Implementations

The prototype MPI implements a subset of MPI-1.2 [1]. With the exception of `MPI_Barrier()`, only basic point-to-point communication and basic support functions were implemented(Figure 4). Only support for basic MPI Datatypes is included and `MPI_COMM_WORLD`, is the only group. The MPI was implemented in roughly 1600 lines of C++ and compiled with GNU g++ 3.3[5].

[5]gcc version 3.3 20030304 (Apple Computer, Inc. build 1495)

`MPI_Comm_rank()`	`MPI_Isend()`
`MPI_Comm_size()`	`MPI_Recv()` †
`MPI_Finalize()`	`MPI_Send()` †
`MPI_Init()` †	`MPI_Wait()`
`MPI_Irecv()`	`MPI_WaitAll()` †
`MPI_Barrier()` †	

Fig. 4. Subset of MPI implemented. †indicates functions which are built from other MPI functions.

The primary data structures are a series of linked lists to contain requests and the state required to advance them.

- `postedRecvQ`: Posted receive buffers for incoming messages to match against.
- `activeRecvQ`: Active receive requests which require processing (i.e. rendezvous requests which must send a reply, requests waiting for a DMA engine, etc.)
- `unexpectedQ`: List of unexpected messages which have arrived. Used by receive to match against.
- `unexpectedActiveQ`: Active unexpected messages which must be advanced (i.e. unexpected messages requiring DMA transfer).
- `sendQ`: Queue message send requests for processing.

All of these primary data structures reside in the NIC memory. Almost all processing occurs on the NIC. The main processor is only required to dispatch message requests to the NIC and wait for request completion.

The NIC continually executes a loop that performs four actions: checking the network for new incoming messages; checking for any new requests from the main processor; advancing active requests; and updating the ALPU. The network is polled for new incoming messages. If a new message is detected, the message headers are stripped off and compared against posted receive buffers. If a match is found, the receive request is moved to the active list so it can set up a DMA or send a rendezvous reply. If no match is found, the message is entered on the `unexpectedQ`, to be matched against future receives. Active send requests are advanced by allocating network and DMA resources and performing the send. Once the send is com-

pleted, resources are freed. Receive requests first try to match against the `unexpectedQ` to see if their message has already arrived. If no match is found, they are added the the `postedRecvQ` where they await an incoming header to match. When they are matched, they can perform any required DMA transfers before informing the main processor of their completion.

Use of the ALPU requires minimal modification of this basic structure. Each iteration of the NIC's loop updates the posted receive ALPU and the unexpected ALPU. A pointer is kept to indicate which portions of the `postedRecvQ` and `unexpectedQ` have been transfered to the ALPU and which have not. If there are portions of these lists that have not yet been added to the ALPU, the NIC will attempt to insert them. The NIC sends a `START INSERT` message, and then drains the ALPU's result FIFO of any match results until a `START ACKNOWLEDGE` is received. It then attempts to insert as many of the remaining headers as it can, updating the pointer to indicate which portions of the queue have been inserted. After the inserts, it will send a `STOP INSERT` command.

When new incoming messages arrive, their headers are automatically sent to the ALPU. When the NIC detects these messages, it checks the ALPU's output queue to see if it has matched. If it does, the relevant request is removed from the `postedRecvQ`. If no match is found, the portion of the `postedRecvQ` that is not on the ALPU is checked. If no match is found, the message headers are added to the `unexpectedQ` and will be inserted into the widget.

Similarly, when receive requests arrive, the unexpected ALPU is checked to see if a match has occurred. If the widget returns `MATCH FAILURE`, any portion of the `unexpectedQ` not on the ALPU is checked.

D. FPGA Prototype

To provide a reasonable estimate of the size and operating frequency of the ALPU, a prototype implementation was created, targeting Xilinx Virtex 2 and Virtex 2 Pro FPGAs. The ALPU was designed using JHDL [26], a structural design tool that provides fine-grained control over the placement of logic on the FPGA. The final design is parameterized to allow different match and tag widths, as well as different combinations of the total number of cells and the number of cells in each block.

When designing the unit, the top priorities were small area, high speed and regularity in placement. The regular placement constraint arose from the need to create a placement scheme that was programatically adaptable to different combinations of match and tag widths. To allow for higher operating frequencies, the ALPU has been pipelined. The pipelining used in the design does not allow execution overlap, and the final implementations can process a new match every 6 or 7 clock cycles (depending on the total number of cells in the ALPU and the block size). The pipeline stages are broken down as follows:

1) Fanout global signals to the blocks of cells; each block registers its own copy of these signals.
2) Produce a match or not match for each cell.
3) Perform the priority muxing within each block.
4) Perform the prioriry muxing between blocks to determine if there was a match. If a match was found, this stage also produces the matched tag and the address of the highest priority cell that matched. This stage is either one or two cycles, depending on the circuit parameters.
5) Fanout the delete signals. Again, each block registers its own copy of these signals.
6) Delete the matched cell.

If desired, it is possible to overlap execution of the first and last stages (i.e., new match data can be distributed to the blocks while the last match is being deleted). The simulation results assume a 7 cycle pipelining latency with no overlap of execution. The current pipelining scheme also allows inserts to happen on every other clock cycle.

VI. Results

Three sets of experiments were performed. The first was an FPGA-based prototype used to explore size and performance issues of the design. The second experiment simulated the performance of a NIC with and without the associative list processing unit (ALPU) for the posted receive queue. In the

final experiment, the ALPU was applied to the management of the unexpected message queue. Results from these experiments indicate that the ALPU is small and fast enough, and provides sufficient benefits to be practical.

A. FPGA Prototype

This section details the sizes and speeds of the ALPU prototypes. Prototypes for list units accelerating both posted receives and unexpected messages were created. The Xilinx FPGA tool chain was used to map the prototypes to a Virtex-II Pro 100 FPGA with a -5 speed grade[6]. We chose to test units with both 256 and 128 total cells, with block sizes of 8, 16, and 32. For each test, the match width was set to 42 and the tag width was 16. These widths are adequate to support an MPI implementation supporting the full specification on a 32K node system. In addition, there is a mask bit for every match bit[7].

The sizes and speeds of the prototypes are found in Tables IV and V. The size and speed numbers were taken from the reports generated by the Xilinx tools. The sizes include the number of 4-input lookup tables (LUTs), the number of flip flops (FFs), as well as the number of slices[8]. The speeds were obtained by constraining the clock to 9ns. Therefore, the prototypes with block sizes of 8 and 16 will likely run at even higher frequencies.

TABLE IV
SIZES AND SPEEDS OF THE POSTED RECEIVES ALPU PROTOTYPES.

Total Cells	Block Size	Size LUTs	Size FFs	Size Slices	Speed (MHz)	Latency
256	8	17,372	28,908	15,766	112.5	7
	16	17,573	27,656	15,090	111.4	7
	32	18,054	26,971	14,742	100.2	6
128	8	8,687	14,562	7,945	111.5	7
	16	8,786	13,897	7,606	112.1	6
	32	9.025	13,605	7,431	100.6	6

[6]This is a 0.13 micron design. For reference, two faster speed grades are currently available on the same process technology.

[7]Providing a mask bit for every match bit allows maximum configurability and supports protocols beyond MPI, such as Portals. Thus, this configuration is the "worst case" size and speed for a real implementation.

[8]A slice consists of two LUTs and two FFs and a small number of dedicated logic units, but frequently cannot be used this densely.

TABLE V
SIZES AND SPEEDS OF THE UNEXPECTED MESSAGES ALPU PROTOTYPES.

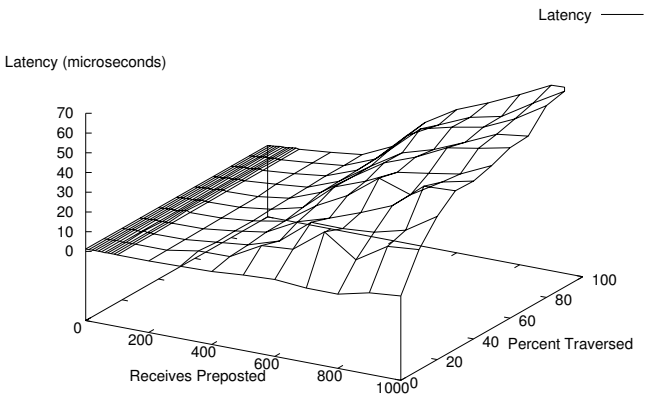
Total Cells	Block Size	Size LUTs	Size FFs	Size Slices	Speed (MHz)	Latency
256	8	17,339	19,414	11,562	112.1	7
	16	17,556	17,490	10,631	111.9	7
	32	18,045	16,469	10,350	100.9	6
128	8	8,672	9,773	5,806	111.2	7
	16	8,777	8,771	5,356	112.1	6
	32	9,020	8,311	5,215	100.6	6

Though the ALPU is quite large in an FPGA (the 256-entry posted receive ALPUs consume approximately 35% of the FPGA), as an ASIC, the size would be similar to that of commercially available ternary CAMs. We also estimate that the move to standard cell ASIC technology would provide a $5\times$[9] increase in clock frequency. This means that the prototypes would all run at about 500MHz; this is equivalent to the core logic speed in the ASC Red Storm network interface[5]. The implied size and speed of the ALPU in an ASIC makes it a good candidate for addition into a network interface offload engine.

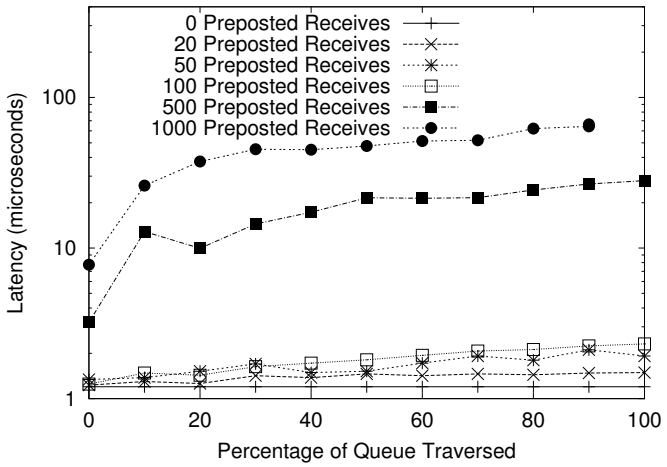
B. Preposted Latency Impacts

Figure 5 compares the performance of a baseline NIC (similar in nature to what will be in the Red Storm system) to the same NIC enhanced with a 128-entry ALPU and a 256-entry ALPU. On the left, the full 3D surface is shown for each configuration while the right shows projections of several of the lines on a 2D graph. The graphs have some interesting traits. For the baseline NIC (parts (a) and (b)), the low end of the graph shows each entry traversed adding an average of 15 ns of latency. By comparison, for a Quadrics Elan4 NIC, each entry traversed adds 150 ns of latency. The $10\times$ performance improvement is not surprising because the NIC being modeled has a significantly faster clock ($2.5\times$), is dual issue (for integers, floating-point does not get used), and has separate 32 KB instruction and data caches. When the queue is too long to fit in cache, the average time per entry traversed grows to 64 ns. This overhead shows

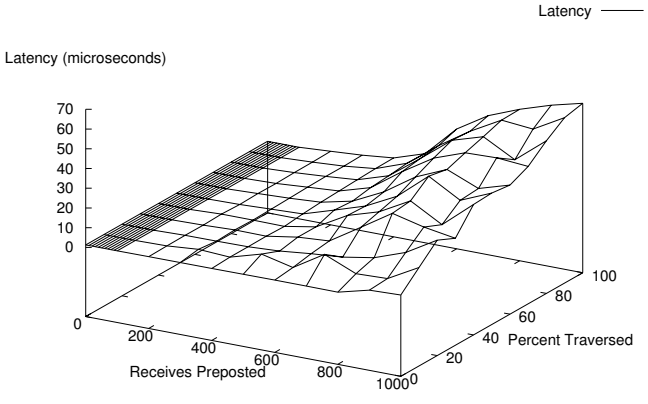
[9]A $5\times$ increase from FPGA to standard cell ASIC is an extremely conservative estimate. It would likely be larger.



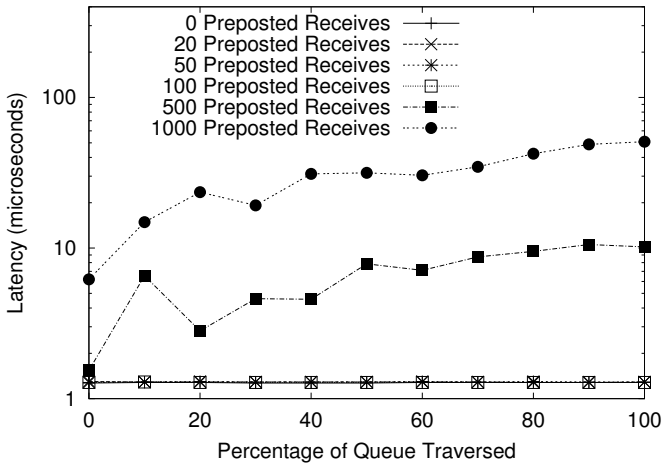
(a)



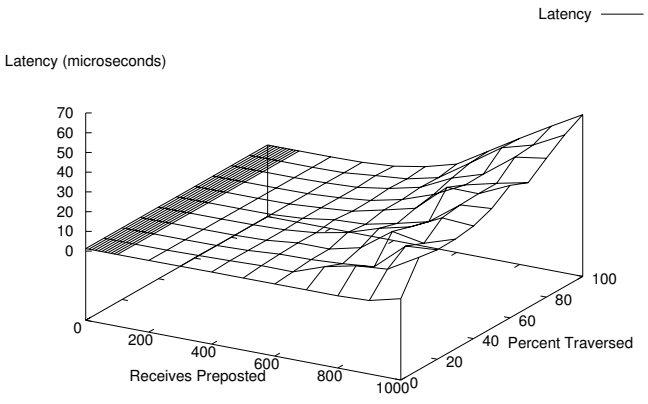
(b)



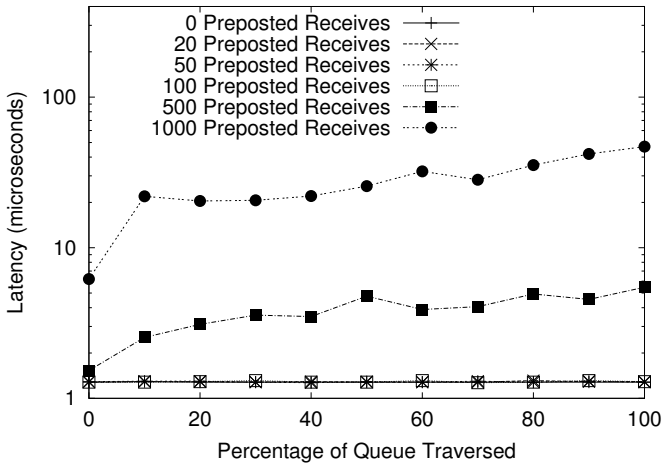
(c)



(d)



(e)



(f)

Fig. 5. **(a) & (b)** Growth of latency with standard posted receive queue; **(c) & (d)** Growth of latency using a 128-entry ALPU to manage posted receive queue; **(e) & (f)**Growth of latency using a 256-entry ALPU to manage posted receive queue

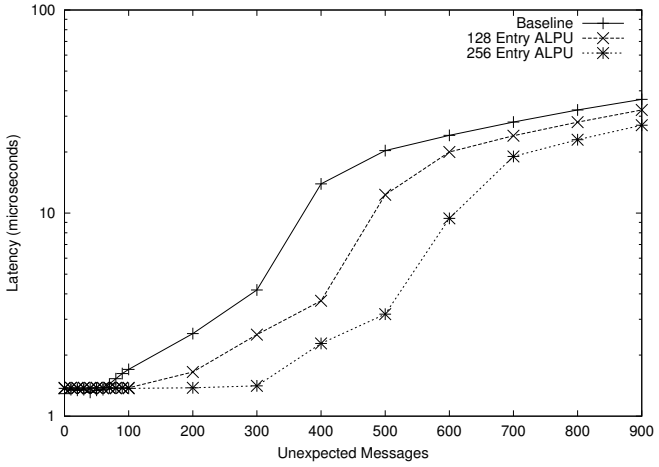


Fig. 6. Growth of latency with unexpected queue length

up even when the entire list is not traversed. For example, the time to traverse an entire 400-entry list is $13\mu s$ and the time to traverse 80% of a 500-entry list is $24\mu s$.

Incorporating an ALPU yields two significant advantages, as shown in Figure 5. The most dramatic advantage is a flat latency curve until the length of the posted receive queue crosses the size of the ALPU. The penalty is an 80 ns increase in the baseline latency (zero-length posted receive queues) as the processor incurs overhead from being forced to interact with the ALPU. With 5 entries in the posted receive queue, the ALPU breaks even. Thus, it is entirely possible that the MPI library could be optimized to not use the ALPU until the list is at least 5 entries long. The second advantage provided by the ALPU is the reduction in the usage of the cache. By using the ALPU, the processor is not required to traverse the first N entries of the queue, even if the ALPU does not find a match. The storage required by the ALPU is relatively small (the entire queue entry does not have to be stored). Each entry in the ALPU contains matching data only, but the processor stores several other pieces of data in the queue entry. Thus, the number of cache lines the processor must retrieve from memory is dramatically reduced if it does not have to search the first several entries.

C. Unexpected Message Impacts

In contrast to the impacts on the preposted queue, the measurements show no advantage from the ALPU for applications that have extremely short unexpected message queues. Indeed, with short unexpected message queues, the ALPU appears to show a small loss in latency performance (a few tens of nanoseconds). As can be seen in Figure 6, after the unexpected queue reaches a length of 70 entries, the ALPU begins to offer a clear and significant advantage. An interesting phenomenon is seen in each line: as the cache on the processor in the NIC is exhausted, the latency rises more dramatically. This mirrors the behavior seen in the management of the preposted queue, and, again, the ALPU is able to delay the point at which this rapid growth in latency occurs.

What is missing in these graphs is the real advantage of the ALPU. The benchmark is written as conservatively as possible while still attempting to demonstrate the limitations of a long unexpected queue. Thus, the time to post a receive is allowed to be overlapped with the time to transfer the messages. In real life, a long posted receive queue is created by pre-posting several receives consecutively (without matches arriving). Each receive would take progressively longer and would impact the application execution time directly. In such a case, the ALPU would offer a much greater benefit.

VII. Conclusions

Both the posted receive queue and the unexpected message queue can be significant bottlenecks in the processing of MPI messages. This paper presents a novel feature to be integrated in a network interface to accelerate the processing of both of these critical queues in MPI. The associative list processing unit (ALPU) was prototyped in an FPGA and was found to be small enough and fast enough to be integrated in a modern network interface.

To assess the performance impact of the proposed accelerator, a system simulator was used to simulate a baseline NIC as well as a NIC enhanced with the proposed feature. The addition of the ALPU was found to add minimal overhead, even when used on extremely short queues. As the queue length grew, the addition of the ALPU demonstrated dramatic drops in impacts of queue length on latency. Even

when the queue length grows beyond the size of the ALPU, the addition of the ALPU is an inexpensive way to decrease the pressure on the cache for the processor in the NIC.

VIII. Future Work

The optimization of MPI is a broad and ongoing effort. Focus areas include other optimzation techniques to further accelerate queue traversal and techniques to traverse queues quickly with fewer hardware resources. Another area of research will focus on how to offload significant portions of the Portals interface to enable support of MPI, run-time software, and I/O.

References

[1] Message Passing Interface Forum, "MPI: A message-passing interface standard," *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, 1994.

[2] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993, pp. 1–12.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Sheiman, "LogGP: Incorporating long messages into the LogP model," *Journal of Parallel and Distributed Computing*, vol. 44, no. 1, pp. 71–79, 1997.

[4] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[5] R. Alverson, "Red Storm," in *Invited Talk, Hot Interconnects 10*, August 2003.

[6] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics network: High-performance clustering technology," *IEEE Micro*, vol. 22, no. 1, pp. 46–57, January/February 2002.

[7] Myricom, Inc., "Myrinet Express (MX): A high performance, low-level, message-passing interface for Myrinet," July 2003. [Online]. Available: http://www.myri.com/scs/MX/doc/mx.pdf

[8] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, Santa Fe, NM, April 2004.

[9] R. Brightwell, S. Goudy, and K. D. Underwood, "A preliminary analysis of the MPI queue characteritics of several applications," *submitted*, May 2004. [Online]. Available: ftp://ftp.cs.sandia.gov/pub/papers/bright/mpi-queue-apps.pdf

[10] K. D. Underwood and R. Brightwell, "The impact of MPI queue usage on message latency," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[12] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386. [Online]. Available: http://www.lam-mpi.org/download/files/lam-papers.tar.gz

[13] R. Dimitrov and A. Skjellum, "An efficient MPI implementation for Virtual Interface (VI) Architecture-enabled cluster computing," in *Proceedings of the Third MPI Developers' and Users' Conference*, March 1999, pp. 15–24.

[14] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting, Linz, Austria*, ser. Lecture Notes in Computer Science, D. Kranzlmuller, P. Kacsuk, J. Dongarra, and J. Volkert, Eds., vol. 2474. Springer-Verlag, September/October 2002.

[15] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski, "A network-failure-tolerant message-passing system for terascale clusters," *International Journal of Parallel Programming*, vol. 31, no. 4, pp. 285–303, August 2003.

[16] P. Shivam, P. Wyckoff, and D. Panda, "EMP: Zero-copy OS-bypass NIC-driven gigabit ethernet message passing," in *Proceedings of the 2001 Conference on Supercomputing*, Nov. 2001.

[17] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen, "Portals 3.0: Protocol building blocks for low overhead communication," in *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[18] D. Buntinas, D. K. Panda, and P. Sadayappan, "Fast NIC-based barrier over Myrinet/GM," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[19] D. Buntinas and D. K. Panda, "NIC-based reduction in Myrinet clusters: Is it beneficial?" in *Proceedings of the SAN-02 Workshop (in conjunction with HPCA)*, February 2002.

[20] A. Moody, J. Fernandez, F. Petrini, and D. K. Panda, "Scalable NIC-based reduction on large-scale clusters," in *Proceedings of the ACM/IEEE SC2003 Conference*, November 2003.

[21] B. Plattner, G. Varghese, J. Turner, and M. Waldvogel, "Scalable high-speed prefix matching," February 2002. [Online]. Available: http://marcel.wanda.ch/Publications/waldvogel01scalable.pdf

[22] R. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen, "The Portals 3.0 message passing interface," Sandia National Laboratories, Tech. Rep. SAND99-2959, December 1999.

[23] R. Brightwell, A. B. Maccabe, and R. Riesen, "Design, implementation, and performance of MPI on Portals 3.0," *International Journal of High Performance Computing Applications*, vol. 17, no. 1, pp. 7–20, Spring 2003.

[24] A. Rodrigues, "Enkidu discrete event simulation framework," University of Notre Dame, Tech. Rep. TR04-14, 2004.

[25] D. Burger and T. Austin, *The SimpleScalar Tool Set, Version 2.0*, SimpleScalar LLC.

[26] B. Hutchings, P. Bellows, J. Hawkins, S. Hemmert, B. Nelson, and M. Rytting, "A CAD suite for high-performance FPGA design," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, K. L. Pocek and J. M. Arnold, Eds., IEEE Computer Society. Napa, CA: IEEE, April 1999, pp. 12–24.